Express Mail No. EM209353178US

Description

METHOD AND SYSTEM FOR CUSTOMIZING FORMS IN AN ELECTRONIC MAIL SYSTEM

Technical Field

This invention relates to an improved method and system for communicating information through electronic mail system and in particular a method and system of using customizable forms in an electronic mail system.

Background of the Invention

It is common in written communications to use standard forms. Examples of standard forms are credit application and phone message slips. These forms allow for the collection of certain data in a structured format. This structured format simplifies the processing of the data.

In computer systems, data is often gathered and displayed through the use of electronic forms. For example, a computer program could display a form that looks similar to the paper version of a credit application form. This similarity simplifies the entry of data into the computer and subsequent display of the data.

An electronic mail system allows mail to be collected electronically through a computer terminal and transmitted to another computer or another user of the same computer and displayed on a terminal. Typical mail systems use certain standard forms. For example, the standard send message form 100 as shown in Figure 1 has a to field 101 into which the user enters the recipient of the mail messages and a re field 102 into which the user enters the subject of the mail. The send message form 100 also has text field 103 into which the user enters the body of the message. A mail system would typically have an analogous receive message form for displaying a message.

A few mail systems have allowed the user to add custom forms. These mail systems provide only limited customization. The customization is typically limited to the use of predefined components. For example, the forms designer could specify where to place a text field or a date field. However, the mail system predefines how the fields will operate. When a user of a form presses a key or uses a mouse button to click on a component of a form, the mail system will typically take one or more actions in response to that input. The actions taken when a button field is clicked, for example, is referred to as the "behavior" of the button. Each form component in custom forms traditionally has a single behavior or a fixed number of possible predefined behaviors.

While this limited customization allows for some degree of user-customization of electronic mail forms, the user is limited to the predefined components and behaviors.

Summary of the Invention

It is an object of the present invention to provide a method and system for customizing forms in an electronic mail system.

It is another object of the present invention to provide an electronic mail system in which the user of the mail system can specify the field layout of a custom form and specify the behavior of the fields in the custom form.

It is another object of this invention to provide an electronic mail system that collects data through a custom form, packs the data into a mail message, and transports the mail message to the specified recipients of the mail.

It is another object of this invention to provide an electronic mail system that receives mail messages, unpacks the data from the mail message, and displays the data in a custom form.

It is another object of the present invention to provide an electronic mail system with a transaction event processor to receive mail events and to call a form control procedure to implement the behavior of the custom form.

It is another object of the present invention to provide a layout for a form data structure that includes the definition of the form fields and the form control procedure.

Brief Description of the Drawings

Figure 1 shows an example of a typical send message form.

Figure 2 shows the components of a mail system that implements custom forms.

Figure 3 shows an example of a custom form.

Figure 4 shows the layout of the form data structure.

Figure 5 is a flow diagram of the main routine of the TREV.

Figure 6 is a flow diagram of subroutine FCPrequest.

Figure 7 is a flow diagram of subroutine MouseEvent.

Figure 8 is a flow diagram of subroutine CurrentField.

Figure 9 is a flow diagram of subroutine KeyboardEvent.

Figure 10 is a flow diagram of subroutine Enable/Disable.

Figure 11 is a flow diagram of subroutine PackEvent.

Figure 12 is a flow diagram of subroutine FCP.

Figure 13 is a flow diagram of subroutine formNew.

Figure 14 is a flow diagram of subroutine fieldPre.

Detailed Description of the Invention

In a preferred embodiment of the present invention, an electronic mail system implements user-customizable forms that allow the user to define form components and their behavior. This invention allows a user to specify the layout of a custom form and specify a form control procedure (FCP) to control the behavior of the form components. The FCP is a computer subroutine that implements user-defined processing of the form. The form is defined in a form data structure that contains the layout of the form and the FCP. The mail system interacts with the FCP to collect form data and transmit the data to a receiver. The mail system also interacts with the FCP to display the message through the custom form when it is received.

In a preferred embodiment, the mail system has a transaction event manager (TREV) that calls the FCP. The TREV creates a window for the form and displays the form in the window. When certain events occur, such as keyboard entry, for the window, the TREV calls the FCP. This calling allows the FCP to perform custom processing.

Figure 2 shows the components of a mail system that uses custom forms. The mail system 201 contains the TREV 202. The TREV accesses the form data structure 203 to display the form window 204. The mail system 201 receives input from the keyboard 205 and mouse 206. The mail system 201 packs the message data into mail message format and transports the message to the recipient via electronic mail link 207. The mail system 201 also receives mail messages via electronic mail link 207 and unpacks the messages.

Figure 3 shows an example of a custom mail message form. The form 300 is designed to handle library requests. The form 300 contains picture 301 that is suggestive of the function, field 302 that is a scroll list in which the user selects the recipient's names, subject field 303 which is a text field, request field 304 which is a scrolling text field, check boxes 305 which select the source, other field 306 which is a text field, radio button fields 307 which select the delivery means, and send button field 308 which allows the user to indicate that the message is to be sent.

Form Data Structure

A custom form is defined in a form data structure. The form data structure describes the design and layout of the form. It describes the initial size of the form and initial placement of the form on the display. It describes the placement of predefined form components, such as buttons and text fields, and the placement and appearance of user-defined form components. The appearance of user-defined form components may be defined as an arbitrary bitmap image. Figure 4 shows the layout of the form data structure of a preferred embodiment. The form comprises four types of elements: form header, field object, form control, and form control procedure. In a preferred embodiment, the elements have variable lengths and have a primary and secondary key to allow fast retrieval of the elements. Alternatively, the FCP could be stored separate data structure.

Form Header

The form header contains information describing the window in which the fields are displayed. The form data structure contains only one form header which is the first entry in the form data structure. The primary key of the form header is "FHDR" and the secondary key is 0. The following data structure, as specified in the "C" programming language, defines a preferred format of form header.

```
typedef struct FormHdr
{
     char            formFlags;
     char            formProcId;
     Rect            formCoords;
     short           formCurField;
     unsigned char   formTitle[];
} FormHdr, *FormHdrPtr
```

The variable formFlags specifies automatic positioning of the window and controls the appearance of the window. The variable formFlags can be set to a combination of values as described in the following. The variable formFlags is set to the value ffNoFlags to indicate that variable formCoords contains the coordinates of the window and variable formProcId contains the style of the window. The variable formFlags is set to the value ffTBCentre to indicate that the window is to be centered vertically. The variable formFlags is set to the value ffLRCentre to indicate that the window is centered horizontally. The variable formFlags is set to the value ffTBLRCentre to indicate that the window is centered both vertically and horizontally. The variable formFlags is set to the value ffAtBottom to indicate that the window is to be placed at the bottom of the screen. The variable formFlags is set to the value ffNoMailIcon to indicate that the mail icon is not to be drawn on the title bar. The variable formFlags is set to the value ffModal to indicate that the window is modal. The variable formFlags is set to the value ffGoAway to indicate that the window has not go away box.

The variable formProcId specifies the style of window. In a preferred embodiment the style can be modeless or modal.

The variable formCoords specifies the screen position and size of the form window. This positioning information can overridden by the setting of variable formFlags.

The variable formCurField is used internally by the TREV to store what field is current.

The variable formTitle is a string that is displayed in the title bar of modeless form windows.

Field Objects

A form consists of a number of fields. Each field object describes the characteristics of a field of the form. For each field there is one field object in the form data structure. The primary key of a field object is "FFLD" and the secondary key is set to a unique identifier for that field; typically, the identifier would be a descriptive of the field. For example, an address field may have a secondary key equal to "ADDR." The following data structure defines a preferred format of the field objects.

```
typedef struct FldHdr
{
     short              type;
     unsigned short attributes;
     Rect               coords;
     unsigned short keyEquiv;
     PackedFont         font;
     Handle             data;
     Handle             private;
     unsigned char  initData[];
} FldHdr, *FldHdrPtr;
```

Sub B2

The variable type indicates the type of the field. The following describes some preferred standard field types. One skilled in the art would know that other standard field types can be defined. The variable type is set to the value fieldStaticText to indicate that the text in the field cannot be edited. The variable type is set to the value fieldEditText to indicate that the text in the field can be edited, for example, subject field 303. The variable type is set to the value fieldHiddenText to

~~indicate that the text in the field can be edited but is not echoed to the display. The variable type is set to the value fieldButton to indicate that the field is a standard button, for example, button field 308. The variable type is set to the value fieldRadioButton to indicate that the field is a standard radio button, for example, radio button fields 307. The variable type is set to the value fieldCheckBox to indicate that the field is a standard check box, for example, check boxes 305. The variable type is set to the value fieldButtonIcon to indicate that the field displays a button in the shape of the specified icon. The variable type is set to the value fieldOptionButton to indicate that the field displays a button with a specified title string. The variable type is set to the value fieldTime to indicate that the field displays the time of day. The variable type is set to the value fieldDate to indicate that the field displays the date. The variable type is set to the value fieldRect to indicate that a rectangle is drawn around the field. The variable type is set to the value fieldPicture to indicate that the field displays the specified picture, for example, picture 301. The variable type is set to the value fieldVariableData to indicate that field contains data that is not displayed on the screen. The variable type is set to the value fieldUser to indicate that the field is a user-defined field.~~

The variable attributes specifies the appearance and behavior of the field. One skilled in the art would know that other values for the variable attributes could be defined to specify different appearances and behaviors. The variable attributes is set to the value attrCanBeCurrent to indicate that the field can be the current field. The current field is the field that receives characters entered by the user. The variable attributes is set to the value attrInform to indicate that the FCP is to be called by the TREV whenever the field is affected by a user event, such as, a mouse click or entry of a character (if the field is current). The variable attributes is set to the value attrPack to indicate that the field contents will be packed into the mail message when a send message request is detected by the mail system, for example, when the user clicks button field 308. The variable attributes is set to the value attrIdle to indicate that the FCP will be called periodically. This periodic calling allows fields, such as a time field, to be updated. The variable attributes is set to the value attrReadOnly to indicate that the field is read only. The variable attributes is set to the value attrDisabled to indicate that the field is drawn differently to indicate that it is disabled (usually grey). The variable attributes is set to the value attrGroup1, attrGroup2, attrGroup3, or attrGroup4 to indicate that the field is in an attribute group. The variable attributes is set to the value attrDependCheckState, attrDependSetState, or attrDependSetNegState to define the functioning of a group. A group of fields allows for the enabling and disabling of fields within the group based on whether other fields in the group contain data. When data is entered or deleted from a field, the TREV checks all the fields in the group that have the variable attributes set to the value attrDependCheckState set. If all these fields have data, then the TREV enables all the fields in the group that have the variable attributes set to attrDependSetState and disables all the fields in the group that have the variable attributes set to attrDependSetNegState.

The variable coords contains the coordinates of the rectangle that defines the field in the window.

The variable keyEquiv contains the keyboard equivalent for many types of fields, such as a button field.

The variable font contains the font, face, and size of the text that is displayed in the field.

The variables data and private are handles that are used at run time to store information about the field.

Form Control

The form control fields are a special class of fields that are not referenced or modified by the TREV. The field serves to store global variables for an FCP. The variables are preserved between calls to the FCP. The primary key is "FDAT" and the secondary key is a unique identifier for the field.

Form Control Procedure

Sub B3

The Form Control Procedure (FCP) is a computer subroutine routine that is called directly by the TREV. The FCP written by the forms designer to implement form customization. The FCP is a block of code stored as the last entry in the form data structure. In a preferred embodiment, the FCP is written in assembly language or another programming language that is compiled into machine code. Alternatively, the FCP can be written in a scripting language or pseudo-machine language that is interpreted. The use of a scripting language or pseudo-machine language would facilitate platform independent custom forms. In a preferred embodiment, the FCP has full access to the computer resources. Alternatively, the FCP could be restricted as to the resourced used. For example, the FCP could be restricted to the operating system calls available to it. The primary key of the FCP is "FFCP" and the secondary key is zero. The following defines the format of the call to the FCP.

```
pascal OSErr FCP (UpCall, callType, win, ident, req,
arg)

        ProcPtr    UpCall;
        short      callType;
        WindowPtr  win;
```

```
long      ident;
short     req;
long      arg;
```

The parameter UpCall is an address of a procedure that can be called by the FCP. This parameter provides a convenient mechanism for allowing the FCP to access the internal functions of the mail system. For example, the mail system may support a field type defined as a list, for example, to field 302. The mail system may have routines, such as an add-item-to-list routine, to manipulate lists. The FCP can access these routines through the procedure pointed to by the parameter UpCall.

The parameter callType specifies what particular action the FCP is asked to perform. The parameter callType can be set to the following values: fcpFormEvent, fcpFieldPre, fcpFieldPost, or fcpUserField.

The parameter callType is set to the value fcpFormEvent to allow the FCP to deal with form-specific events. A form-specific event would be global initialization of FCP internal variables. When the parameter callType is set to fcpFormEvent the parameter req specifies the type of form event. The parameter req is set to the value formNew to permit the FCP to initialize its global data and allocate any other data structures it may require. The FCP is called with this parameter value after the individual fields have been initialized. The parameter req is set to the value formDispose to permit the FCP to dispose of any memory manager data structures that have been allocated. The FCP is called with this parameter value before the individual fields have their associated dispose functions performed. The parameter req is set to the value formIdle to permit the FCP to perform idle processing, such as updating a time field. The FCP is called with this parameter value before the idle messages are sent to the individual fields. The parameter req is set to the value formPack to notify the FCP that the individual fields have been packed into the mail message. The FCP can change the packed data or add additional data to the mail message. The parameter req is set to the value formUnpack to notify the FCP that the individual fields have been unpacked. The FCP can modify the unpacked data.

The parameter callType is set to the value fcpFieldPre to allow the FCP to perform customization before the TREV performs its standard functions for an event, such as keyboard entry. When the FCP returns to the TREV, the FCP can set the result code to the value TErrDealtWith to indicate that the TREV is to skip its standard processing for this event.

The parameter callType is set to the value fcpFieldPost to allow the FCP to perform customization after the FCP performs it standard functions for an event.

The parameter callType is set to the value fcpUserField to allow the FCP to perform customization for a user-defined field.

The parameter win contains a pointer to the window in which the form is displayed.

The parameter ident contains the identification of the field, which in a preferred embodiment is the secondary key from the form data structure.

The parameter req contains information on the type of event for which the FCP is being called. One skilled in art would know that other event types, such as list processing events, could be defined. The FCP is called with the parameter req set to the value reqCreate once for each field after a form is created. This allows the FCP to perform initialization associated with the field. The FCP is called with the parameter req set to the value reqDelete once for each field just before the form is disposed of. This allows the FCP to perform clean up for the field. The FCP is called with the parameter req set to the value reqUpdate for each field that needs to have its contents updated on the display. This allows the FCP to redisplay data after the window has been uncovered. The FCP is called with the parameter req set to the value reqIdle periodically for each field with the variable attributes set to the value attrIdle. This allows the FCP to update fields, such as a time field. The FCP is called with the parameter req set to the value reqCurrent for a field that has been tabbed to or clicked upon. This allows the FCP to customize a field when it becomes current. The FCP is called with the parameter req set to the value reqNotCurrent when the current field changes. The FCP is called with the parameter req set to the value reqEnable to indicate that the field has become enabled. The FCP is called with the parameter req set to the value reqDisable to indicate that the field has become disabled. The FCP is called with the parameter req set to the value reqKey to indicated that a key has been entered into the field. The FCP is called with the parameter req set to the value reqChosen to indicate that the defined equivalent key as stored in variable keyEquiv for the field has been entered. The FCP is called with the parameter req set to the value reqMouse whenever a mouse down event (click) occurs. The FCP is also passed the location of the cursor. The FCP is called with the parameter req set to the value reqEdit to indicates that an edit function, such as undo, cut, copy, paste, or clear, is requested for the field. The FCP is called with the parameter req set to the value reqHasData so that the FCP can return a value of true if the field has data and false otherwise. The FCP is called with the parameter req set to the value reqGetData so that the FCP can return the value of the data in the field. The FCP is called with the parameter req set to the value reqSetData so that the FCP can change the data in the field.

The parameter arg stores request-specific data.

Transaction Event Manager

The Transaction Event Manager (TREV) is the portion of the mail system that manages the events associated with a form. Figures 5 through 11 are a flow diagram of the TREV. Figure 5 is a flow diagram of the main TREV routine. This main routine creates a window for a form and then waits for events, such as, the click of a mouse or keyboard entry. When an event occurs, this routine determines the event type and calls the appropriate routine to process the event. In block 501, the routine performs the necessary interaction with the window manager to create a window for a form. The routine initializes the window in accordance with the specifications in the form data structure. In block 502, the routine calls subroutine FCP with the values fcpFormEvent and formNew to indicate that the form was just created. This call allows the FCP to perform customized initialization. Subroutine FCP is described below in detail. In block 503, the routine calls subroutine FCPrequest with the value reqCreate once for each field in the form data structure. These calls allow the FCP to perform custom initialization for each field. Subroutine FCPrequest is described below in detail.

In blocks 504 through 519, the routine waits for an event to occur, determines the event type, and calls the appropriate routines to process the event. In block 504, the routine waits until an event occurs. In block 505, if the event type is idle, then the routine continues at block 506 to process the idle event, else the routine continues at block 508. In block 506, the routine calls subroutine FCP with the values fcpFormEvent and formIdle to indicate that an idle event occurred. In block 507, the routine calls subroutine FCPrequest with the value reqIdle once for each field in the form data structure that has its attribute set to attrIdle. The routine then loops to block 504 to wait for the next event.

In block 508, if the event type is mouse, then the routine continues at block 509, else the routine continues at block 510. In block 509, the routine calls subroutine MouseEvent to process the mouse event. Subroutine MouseEvent is described in detail below. The routine then loops to block 504 to wait for the next event.

In block 510, if the event type is keyboard, then the routine continues at block 511, else the routine continues at block 512. In block 511, the routine calls subroutine KeyboardEvent to process the keyboard event. Subroutine KeyboardEvent is described in detail below. The routine then loops to block 504 to wait for the next event.

In block 512, if the event type is update, then the routine continues at block 513, else the routine continues at block 514. In block 513, the routine calls subroutine FCPrequest with the value reqUpdate once for each field in the form data structure. The routine then loops to block 504 to wait for the next event.

In block 514, if the event type is pack or unpack, then the routine continues at block 515 to process the pack or unpack, else the routine continues at block 517. In block 515, the routine calls subroutine PackEvent to pack or unpack a mail message. Subroutine PackEvent is described in detail below. In block 516, the routine calls subroutine FCP with value fcpFormEvent and value formPack or formUnpack, depending on the event type, to allow the FCP to modify the mail message data. The routine then loops to block 504 to wait for the next event.

In block 517, if the event type is edit, then the routine continues at block 518 to process the event, else the routine continues at block 519. In block 518, the routine calls subroutine FCPrequest with values formCurField from the form data structure and reqEdit. The routine then loops to block 504 to wait for the next event.

In block 519, if the event type is dispose, then the routine continues at block 520, else the routine loops to block 504 to wait for the next event. In block 520, the routine calls subroutine FCP with values fcpFormEvent and formDispose. In block 520, the routine calls subroutine FCPrequest with the value reqDispose once for each field in the form data structure. The routine then returns.

Figure 6 is a flow diagram of the FCPrequest subroutine. This subroutine controls calling the FCP before and after the standard processing is performed by the TREV. The parameters passed to this subroutine depend upon the request type, but typically include the field identification and data. In block 601, the routine calls subroutine FCP with the value fcpFieldPre, the passed request value, the field identification, and the data. This call allows the FCP to perform custom processing for the field. In block 602, if the FCP sets the result code to the value TErrDealtWith, then the routine continues at block 606, else the routine continues at block 603 to perform the standard processing for a field. In block 603, if the field type is UserField, then the routine continues at block 605, else the routine continues at block 604. In block 604, the routine performs the standard process for a field. For example, if the field is an editable text field and the event was the keyboard entry of a letter, then the routine would echo the letter to the display. The routine continues at block 606. In block 605, the routine calls subroutine FCP with the value fcpUserField and the field identification. Since there is no standard processing for a user-defined field, the TREV lets the FCP perform custom processing. The routine then continues at block 606. In block 606, the routine calls subroutine FCP with the value fcpFieldPost, the passed request value, the field identification, and the data. The routine then returns.

Figure 7 is a flow diagram of subroutine MouseEvent, which processes mouse events. The routine is passed the type of mouse event, such as click down, and the location of the cursor. In block 701, the routine determines at what field the cursor is located. In block 702, if the attribute for that field is attrCanBeCurrent, then the routine continues at block 703, else the routine continues at block 704. In block 703, the routine calls subroutine CurrentField to change the current field to the field just selected by the mouse event. Subroutine CurrentField is described in detail below. In block 704, the routine calls subroutine FCPrequest with the value reqMouse, the location, and the field identification. The routine then returns.

Figure 8 is a flow diagram of the subroutine CurrentField, which switches the current field to the passed field. In block 801, the routine calls subroutine FCPrequest with value reqNotCurrent and the formCurField from the form data structure. This call switches the current field to a not current status. In block 802, the routine sets formCurField in the form data structure to the passed field. In block 803, the routine calls subroutine FCPrequest with values reqCurrent and the formCurField from the data structure. This call switches the new current field to a current status. The routine then returns.

Figure 9 is a flow diagram of the subroutine KeyboardEvent, which processes keyboard events. This routine is passed the keyboard event type. In block 901, if the event is a tab key, then the routine continues at block 902, else the routine continues at block 904. In block 902, the routine determines the next field in the form data structure that has the attribute attrCanBeCurrent. In block 903, the routine calls subroutine FieldCurrent to set the next field to the current field and the routine then returns. In block 904, routine determines if the keyboard event corresponds to a equivalence key as defined for a field in the form data structure. If an equivalence is found, then the routine continues at block 905, else the routine continues at block 906. In block 905, the routine calls subroutine FCPrequest with the value reqChosen and the field for which the equivalence was found and then returns. In block 906, if the current field is in a group, then the routine continues at block 907, else the routine continues at block 908. In block 907, the routine calls subroutine Enable/Disable to enable or disable the fields in the group as appropriate. Subroutine Enable/Disable is described in detail below. In block 908, the routine calls subroutine FCPrequest with the value reqKey. The routine then returns.

Figure 10 is a flow diagram of the subroutine Enable/Disable, which enables or disables the fields in a group. In block 1001, the routine, starting with the first field in the form data structure, selects the next field. In block 1002, if all the fields have been selected, then the routine continues at block 1008, else the routine continues at block 1003. In blocks 1003 through 1007, the routine determines what fields are in the group and whether the dependencies are satisfied. In block 1003, if the selected field is in the group, then the routine continues at block 1004, else the routine loops to block 1001 to select the next field. In block 1004, the routine maintains a list of the fields in the group. In block 1005, if the attribute of the selected field is attrDependCheckState, then the routine continues at block 1006, else the routine loops to block 1001 to select the next field. In block 1006, the routine calls subroutine FCPrequest with the value reqHasData. In block 1007, if the field has data, then the routine loops to block 1001 to select the next field, else the routine returns because the dependency failed. In blocks 1008 through 1013, since the dependency is satisfied, the routine enables or disables the fields in the group. In block 1008, the routine, starting with the first field in the saved list of fields (which comprises the group), selects the next field in the group. In block 1009, if there are more fields in the list, then the routine continues at block 1010, else the routine returns. In block 1010, if the attribute for the selected field is attrDependSetState, then the routine continues at block 1011, else the routine continues at block 1012. In block 1011, the routine calls subroutine FCPrequest with the value reqEnable to enable the field and then the routine loops to block 1008 to select the next field in the group. In block 1012, if the attribute for the selected field is attrDependSetNegState, then the routine continues at block 1013, else the routine loops to block 1008 to select the next field in the group. In block 1013, the routine calls subroutine FCPrequest with the value reqDisable to disable the field and the routine loops to block 1008 to select the next field in the group.

Figure 11 is a flow diagram of subroutine PackEvent, which packs or unpacks the form data into or from a mail message. In block 1101 if the event type is a pack, then the routine continues at block 1102, else the event type is an unpack and the routine continues at block 1105. In block 1102 through 1104, the routine packs the form data into a mail message. In block 1102, the routine calls subroutine FCPrequest with the value reqGetData once for each field in the form data structure with the attribute of attrPack. In block 1103, the routine packs the data that is returned from the request to get data into a mail message. In block 1104, the routine calls subroutine FCP with value formPack. This call allows the FCP to modify the packed mail message. The routine then returns. In block 1105 through 1107, the routine unpacks the data in the mail message. In block 1105, the routine unpacks the data from the mail message and stores the data in the form data structure. In block 1106, the routine calls subroutine FCPrequest with the value reqSetData once for each field in the form data structure with the attribute of attrPack. In block 1107, the routine calls subroutine FCP with the value formUnPack to allow the FCP an opportunity to modify the unpacked data. The routine then returns.

Form Control Procedure

Figure 12 is a flow diagram of the main routine of a typical FCP. This flow diagram shows the procedure for determining the callType in blocks 1201 through 1204, and determining the form events in blocks 1208 through 1212. In blocks 1205 through 1207 and blocks 1212 through 1217, the FCP calls the subroutines to process the events. The subroutines implement the customization of the form.

Sub B4

~~Figures 13 and 14 are an example of a flow diagram of the routine to process the events for an FCP that implements a game of tick-tack-toe. Figure 12 is the flow diagram for the main routine in the FCP for this example. The customized form in this example works as follows. There are nine fields in the form. Each field corresponds to a location in tick-tack-toe grid. The fields contain either no data, an X, or an O. The first player would click the mouse over one of fields. The FCP detects that this field becomes the current field, draws an X in the field, and sets the data in the field to an X. The first player would request that the mail system send the message to the second player. When the message arrives at the second player, the FCP would draw the tick-tack-toe grid and draw an X in the appropriate field. The second player would move the cursor to a field and click the mouse. The FCP would draw an O in that field. The second player would then send the message to the first player. Play would continue until one of the players win or all the fields contain an X or an O (a tie). When a player wins, the FCP draws a line through the winning fields and prohibits the placement of any other X's or~~

08938458.050697

O's. The FCP also prohibits placing an X or an O in a field that is already occupied.

Sub B5 Figures 13 and 14 show the flow diagrams for subroutines formNew and fieldPre that are called by the main FCP routine shown in Figure 12. This example of tick-tack-toe using customizable form could be made more sophisticated, for example, by allowing a player to change his mind before the mail is sent. Figure 13 shows the flow diagram for subroutine formNew. The only function of this routine is to draw the tick-tack-toe grid after the window is created.

Figure 14 shows the diagram for the subroutine fieldPre. The subroutine performs all the substantive processing for the implementation of the game. In block 1401, if the value in the parameter req is reqUnPack, then the routine initializes the grid with the current value of the field by continuing at block 1402, else the routine continues at block 1406. In blocks 1402 through 1405, the routine draws an X or an O in the field that is being unpacked, as appropriate. In block 1402, if the data in the field is an X, then the routine continues at block 1403, else the routine continues at block 1404. In block 1403, the routine draws an X in the field and continues at block 1413. In block 1404, if the data in the field is an O, then the routine continues at block 1405, else the routine continues at block 1413. In block 1405, the routine draws an O in the field and continues at block 1413.

In block 1406, if the variable winner is set, then the routine continues at block 1407, else the routine continues at block 1408. In block 1407, the routine sets the result code to the value TErrDealtWith to indicate that the TREV will not need to perform its standard processing for this event. The routine then returns.

In block 1408, if the value of parameter req is reqCurrent, then the routine continues at block 1409, else the routine returns. In block 1409, if data is in the field, then the routine returns because an X or an O is already in the field, else the routine continues at block 1410. In block 1410, if there are an even number of fields with data, then it is X's turn and the routine continues at block 1412, else it is O's turn and the routine continues at block 1411. In block 1411, the routine draws an O in the field and sets the field data value to indicate an O. In block 1412, the routine draws an X in the field and sets the field data value to indicate an X.

In block 1413, the routine determines if there is a winner and if so, then the routine continues at block 1414, else the routine continues at block 1415. In block 1414, the routine draws a line through the grid to indicate the winning fields and set the variable winner. In block 1415, the routine sets the result code to the value TErrDealtWith to indicate that the TREV will not need to perform its standard processing for this event. The routine then returns.

In a preferred embodiment, each user of the electronic mail system has access to the custom forms. In a local area network, the data structure for a form could be stored on a file server. Each user who sends and receives messages using a custom form would download the form from the file server to create or view the message. The mail message that is sent would specify the custom form associated with the message. Alternatively, the custom form could be sent to the recipient as part of the mail message. In a preferred embodiment, each custom form has an associated icon. The mail system would display the icon to indicate that the custom form is available to the user.

It will be appreciated that, although specific embodiments of the invention have been described herein for purposes of illustration, various modifications may be made without departing from the spirit and scope of the invention. Accordingly, the invention is not limited except as by the appended claims.

pm\c\6695409\MacMail\v3